Safe and Atomic Run-time Code Evolution for Java and its Application to Dynamic AOP

Rafał Hryciuk

Based on paper “Safe and Atomic Run-time Code Evolution for Java and its Application to Dynamic AOP” by Thomas Wurthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, Hanspeter Mossenbock

March 19, 2012

Table of Contents

Run - time code evolution

Run - time code evolution

Run-time code evolution allows to change the semantics of a running program. The program is temporarily suspended, parts are replaced with new code, and then the execution continues with the new version of the program.

Code updates use cases

Code updates use cases

- Development and debugging. Eliminates necessity of restarting application.
- Long running servers. Reduces the downtime during migration of the application to a new version.
- AOP tools that perform aspect weaving at run time.

Virtual Machine Layer

Virtual Machine Layer

Using a virtual machine (VM) to execute programs helps solving the challenges of code evolution. The VM increases the possibilities for dynamic code evolution because of the additional abstraction layer between the executing program and the hardware. The main tasks of this intermediate layer are automatic memory management, dynamic class loading, and code verification. The algorithms for code evolution presented in this presentation make heavy use of the existing VM infrastructure.

Java HotSpot ^{TM} VM to Dynamic Code Evolution VM (DCE VM)

Java HotSpot^{TM} VM

Dynamic class redefinition replaces a set of loaded classes with new versions. The current product version of the Java HotSpot^{TM} VM allows such changes as long as only the bodies of methods are affected.

Dynamic Code Evolution VM (DCE VM)

DCE VM allows changes to class member definitions, i.e., adding and removing methods and fields. Additionally, it allows changes to the class hierarchy such as changing the super class or the set of implemented interfaces.

Sketch of the algorithm

Sketch of the algorithm

- The VM searches for subclasses of redefined classes as they can be affected by the change.
- All Java threads pause at safepoints.
- Algorithm scans the heap in order to update pointers to the old class version to become pointers to the new class version.
- Modified methods that are active on the stack at the time of redefinition continue in the old version of their bytecodes. It is only guaranteed that subsequent calls to methods target the latest version of a method.

Limitations and Motivation for Improvement

Development and debugging

The continued execution of an old method can lead to invocations of old deleted methods or to accesses of old deleted fields. When the code evolution is used to speed up the development process by reducing the number of necessary program restarts, this is acceptable. The worst case scenario is that the programmer has to restart the application under development, which would anyway be necessary without the enhanced VM.

Long-running server applications and AOP

The requirements in this case are significantly higher regarding stability and correctness of class updates. Changes to Java programs that can impair the correctness of currently loaded bytecodes are called binary incompatible changes. Such changes may lead to runtime exceptions or even crash the VM.

Limitations and Motivation for Improvement

Binary incompatible changes

- **Deleting Class Members** A method or field that is deleted in the new version of the program may still be accessed from bytecodes of old methods. In such a case, our VM throws a NoSuchFieldError or a NoSuchMethodError, respectively.
- **Type Narrowing** When the subtype relationship between two types A and B is no longer valid in the new version of the program, type safety may be compromised. It can no longer be guaranteed that the dynamic type of a field or local variable is a subtype of its static type. This can lead to a crash of the VM.

Safe update regions

Safe update regions

The extended DCE VM guarantees that the version change is performed immediately if all threads are in safe update regions. Thus, old code is never executed after a successful class update.

Switching to the Extended Program

Base Program

```
class Receiver {

  void recv(BufferedReader in) {
    while(true) {
      String s = in.readLine();



      if (s.length() == 0)
        break;
    }
  }
}
```

Extended Program

```
1  class Receiver {
2    private int count;
3    void recv(BufferedReader in) {
4      while(true) {
5        String s = in.readLine();
6        count++;
7        System.out.println(count);
8        if (s.length() == 0)
9          break;
10     }
11   }
12 }
```

Class Definition Changes

Class Definition Changes

Allowed:

- adding class members
- referring to new classes that are not used in the base program
- replacing an abstract base method with a concrete implementation in the extended version
- adding an interface to a class

Class Definition Changes

Class Definition Changes

Prohibited:

- changing the signature of a field or method
- changing modifiers `static`, `synchronized` and `native`
- replacing concrete implementation with an abstract method
- overriding a non-abstract base method.

Method Body Changes

Restrictions on control flow modifications

- **Branch instructions** in extended code regions must always target bytecodes in the same code region. Therefore, an extended code region cannot be exited with a branch instruction.
- **Return:** The extended code sections may contain a return instruction. This is the only way how parts of the base program can be skipped.

Method Body Changes

Restrictions on control flow modifications

- **Branches:** Exception Handlers: An extended code region must catch any exception that it throws. It is allowed to add exception table entries that cover a range within an extended code region. The exception handler block must however be within the same code region.
- **Exception Interception:** An extended code region may intercept an exception thrown by a base program bytecode. However, it must re-throw the same exception again. The try statement itself does not produce a bytecode, therefore the extended code section that intercepts the exception is continuous to the base code section at the bytecode level.

Method Body Changes

Restrictions on stack frame modifications

- **Operand Stack:** The operand stack height upon exit of an extended code region must be the same as the operand stack height at the entry of the region. Additionally, all values on the stack at the entry of an extended code region must remain unmodified.
- **Local Variables:** The extended code region may introduce new local variables and also read from and write to all local variables of the base program.

Cross Verification

Cross Verification Motivation

As the version change can happen at any bytecode position of the base program, the extended code regions cannot assume that any other extended code region was executed before them. Therefore, an extended code region must not rely on the initialization of local variables in other extended code regions.

Cross Verification

Base Method		Intermediate		**Extended Method**
iconst_1	1	iconst_1	1	iconst_1
istore_1	2	istore_1	2	istore_1
	3		3	**fconst_1**
	4		4	**fstore_1**
iconst_0	5	iconst_0	5	iconst_0
	6	**fload_1**	6	**fload_1**
	7	**fstore_2**	7	**fstore_2**
return	8	return	8	return

Figure 3. Example where cross verification detects an illegal intermediate version.

Transformer Methods

Transformer Methods Motivation

Fields added in the extended version of the program are by default initialized with 0, false, or null. However, an extended program can require other initialization values for its fields.

Transformer Methods Types

- $transformers
- $staticTransformer

Transformer Methods

```
class Receiver {
  StringBuilder b;
  public Receiver() { b = new StringBuilder(); }
  void $transformer() { b = new StringBuilder(); }
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      b.append(s);
      if (s.length() == 0)
        break;
    }
    System.out.println(b);
  }
}
```

Changing Back to the Base Program

- The VM removes fields and methods that are only defined in the extended program.
- The VM guarantees that those deleted members are never accessed after the change.
- The update is delayed until all method activations of all threads are in base code regions (safe update regions) and not in extended code regions.
- There is no guarantee that a safe update region is reached by all threads (timeout).

Safe Changing Back to the Base Program Restrictions

Restrictions

- **Verification**: For a type-safe conversion back to the base method, another kind of cross verification is necessary. The VM performs a modified verification of the base program for each extended code region. It infers the types of local variables at the end of every extended code region. Then, it starts the verifier at the base program bytecode following the code region using the inferred types as the initial types of the local variables. This makes sure that changing from the extended to the base program is a valid operation for each base program position.

Safe Changing Back to the Base Program Restrictions

Restrictions

- **Type Narrowing**: When changing from the base to the extended program, it is allowed to add an interface to a class. When switching back, this would however result in a type narrowing change. Therefore, such a change is only valid if there is no local variable or field violating the subtype relationship between its static and dynamic type.

Safe Changing Back to the Base Program Restrictions

Restrictions

- **State**: The extended code regions must not write to local variables and fields known to the base program. The state of the base program must be read-only for the extended program.
- **Return**: The extended program must not add a return bytecode to a base program method.
- **Call**: Calling a base program method from an extended code region is only allowed if the called method does not modify the state of the base program.

Changing Between Two Arbitrary Programs

AOP Introduction

AOP Introduction

Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects.

AOP concepts

AOP concepts

- **Aspect**: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in J2EE applications.
- **Join point**: a point during the execution of a program, such as the execution of a method or the handling of an exception.
- **Advice**: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.

AOP concepts

AOP concepts

- **Pointcut**: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).
- **Weaving**: linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

The SafeWeave Dynamic AOP System

SafeWeave

- Implemented as a Java agent that is attached to the VM.
- Relies on the DCE VM to allow arbitrary class changes at run time.
- Uses the standard AspectJ weaver (as a black box) to weave aspects.